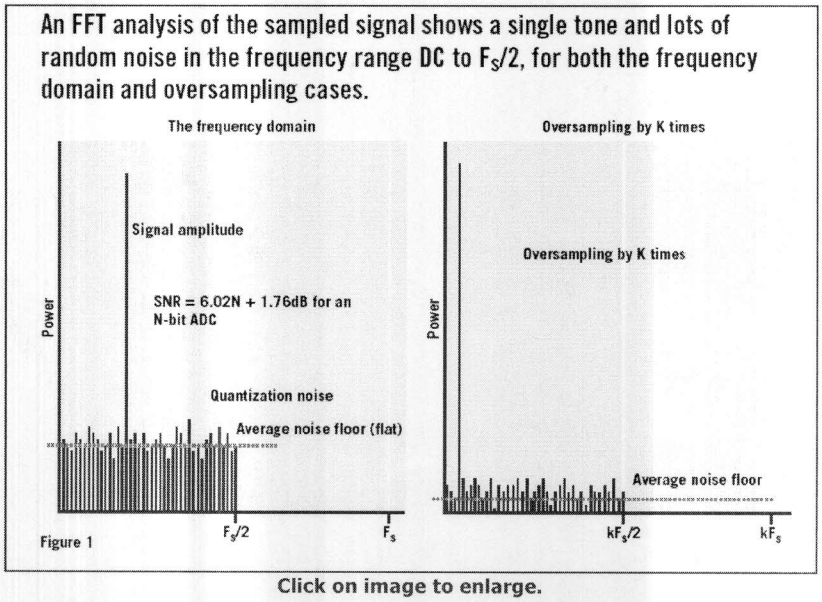
Oversampling with averaging to increase ADC resolution

How an MCU can extend the resolution/accuracy of an ADC by delivering an extra bit or two.

By Franco Contadini, Maxim
Embedded.com
(03/01/10, 12:00:00 AM EST)

When considering the resolution required for an A/D converter (ADC) integrated in a microcontroller (MCU), embedded systems designers must balance cost and performance. Higher ADC resolution implies higher-cost MCUs, but in some cases you can use other features in the MCU to enhance the ADC performance via software. That approach lets you achieve higher resolution using an inexpensive integrated ADC. Here's how to use of oversampling to achieve extra bits of resolution for an ADC integrated in an MCU.

We start by examining the frequency-domain transfer function of a multibit ADC operating on a sinewave input signal. The ADC samples this input at a frequency Fs, which (according to Nyquist theory) must be at least twice the input-signal bandwidth. An FFT analysis (left graph of **Figure 1**) shows a single tone with lots of random noise (known as quantization noise) extending from DC to *Fs*/2.



Click on image to enlarge.

You obtain the signal-to-noise ratio (SNR) by dividing the amplitude of the fundamental by the RMS sum of all frequencies containing noise. For an N-bit ADC whose peak input voltage equals the ADC's full-scale input voltage, the maximum SNR equals 6.02N + 1.76dB.

Consider now the same example, but with sampling frequency increased by an oversampling ratio of *k*, in other words, to *kFs*. An FFT analysis shows that the noise floor has fallen and the SNR is the same, but noise energy has been spread over a wider frequency range (right graph of **Figure 1**).

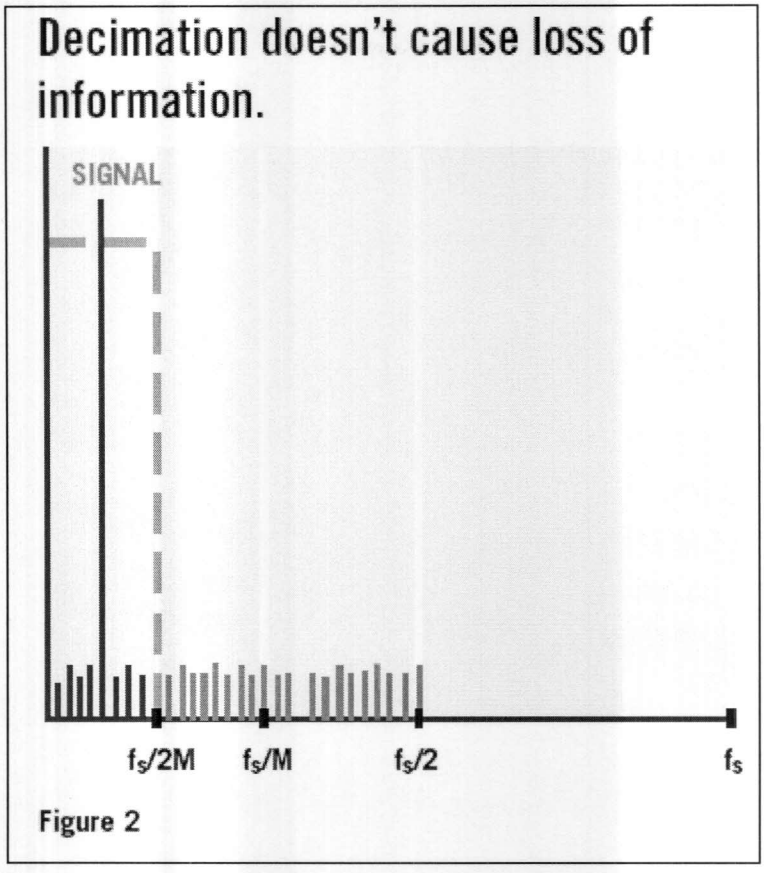


Figure 2

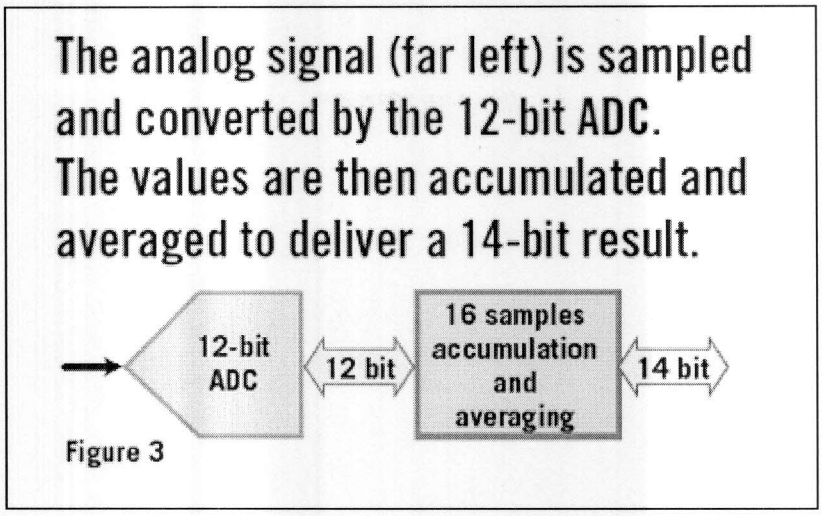
For inputs sampled at F_s, the rate for filtered output data can be reduced to F_s/M without loss of information, using a "decimation" process (**Figure 2**). *M* can have any integer value, on condition that the output data rate is more than twice the signal bandwidth. Oversampling and averaging increases the SNR, which is equivalent to gaining additional bits of resolution. For each such additional bit, the signal must be oversampled by a factor of four:

$f_{os} = 4^w F_s$, where

w is the number of additional bits of resolution desired, F_s is the original sampling frequency required, and f_{os} is the oversampling frequency. As an example, a 12-bit ADC can achieve:

- 13-bit resolution with 4x oversampling (4^1),
- 14-bit resolution with 16x oversampling (4^2),
- 15-bit resolution with 64x oversampling (4^3),
- 16-bit resolution with 256x oversampling (4^4).

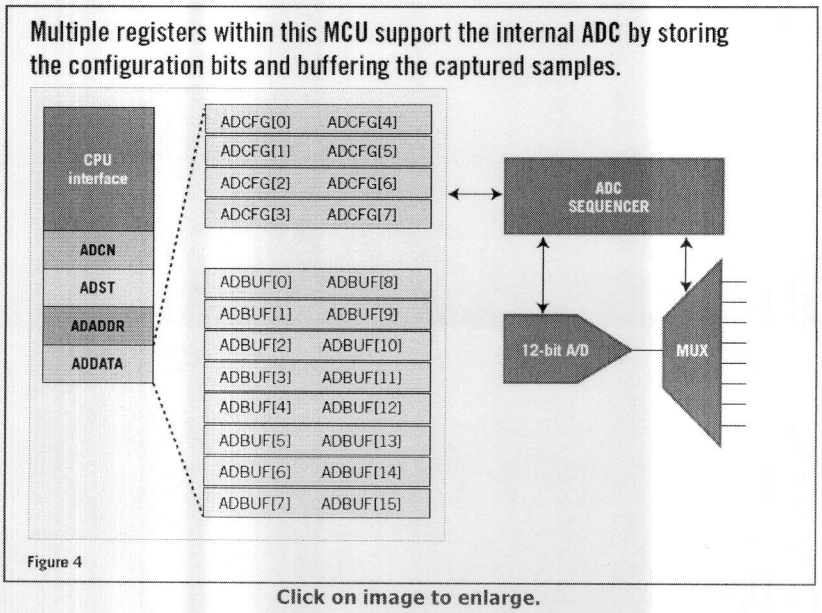
If we oversample an input signal at $16F_s$, we collect enough samples within the required sampling period to average and produce 14 bits of output data, for a 14-bit measurement. This is accomplished by accumulating 16 consecutive samples and dividing the total by 16, as **Figure 3** shows.



This technique requires an ADC with high sampling rate (to increase the resolution without sacrificing the input-signal bandwidth), and an integrated buffer for storing samples. The buffer also helps to reduce the microprocessor overhead. ADCs embedded in MCUs are well suited for this technique, if their integral nonlinearity (INL) and differential nonlinearity (DNL) are in line with the resolution desired.

The oversampling approach can be applied to just about any MCU with an embedded ADC, so to see exactly how it's done, we use a 16-bit RISC-based MCU (Maxim's MAXQ2010) to demonstrate the averaging and oversampling control. The MAXQ2010 incorporates a 12-bit, 312-ksps ADC with 1 LSB of INL and DNL. This MCU (and similar MCUs from other vendors) often include several architectural features that help simplify the oversampling scheme:

- Ability to interrupt on each conversion, each sequence, or on every 12 or 16 samples.
- Capable of taking a single sample, a sequence of samples, or a continuous sequence of samples.
- Inclusion of a 16-word sample buffer.



Click on image to enlarge.

Similarly, most MCUs include registers associated with the ADC that store control parameters for the ADC sequencer and other functions (**Figure 4**). In the MAXQ2010, the following CPU registers control the ADC sequencer:[1]

- **ADCN (ADC Control Register):** Bits in this register control the ADC's sample acquisition extension, power

management override, single/continuous sequence conversion, interrupt intervals, and clock division.

• **ADST (ADC Status Register):** Bits in this register include the register-index selection bits ADCFG (ADC configuration register) and ADBUF (ADC sample buffer register), the conversion start bit, and other status bits for the ADC.

• **ADADDR (ADC Conversion Sequence Address Register):** Bits in this register define the first and last ADCFG registers used in a conversion sequence, as well as the first ADBUF register written in a conversion sequence.

• **ADDATA (ADC Data Register):** Bits in this register serve as read/write access pointers to registers ADCFG[7:0] and ADBUF[15:0].

• **ADCFG[7:0] (ADC Sequence-Configuration Registers):** These eight conversion-configuration registers provide settings for each individual conversion in an ADC conversion sequence. Configuration registers for the start and end of the sequence are given by the SEQSTART and SEQEND bit fields in the ADADDR register.

• **ADBUF[15:0] (ADC Sample Buffer Registers):** A read-only register that contains the ADC conversion result.

Assuming we want to achieve 14 bits of resolution on ADC channel 0, we can use the programming sequence in **Listing 1**. The ADC control register (ADCN) is configured for continuous conversion and for generating an interrupt signal every 16 ADC samples. The bit configuration is shown in **Figure 5a**.

Listing 1 Programming sequence to achieve 14 bits of resolution on ADC channel 0.

```
void initADC()
{
  ADCN = 0xCE0;      // enable continuous conversion, internal reference,
                     // /1 clock frequency, interrupt every 16 samples
  ADST = 0x10;       // select configuration register read/write access on
                     // ADDATA.
  ADDATA = 0x40;     // ADCF [0]AN0 single ended internal reference
  ADADDR = 0x00;     // selecting 0x00 as the first and last conversion
                     // configuration register
  IMR_bit.IM4 = 1;   // set module interrupt enable bit for the interrupt ADC module
}
```

Click on image to enlarge.

A. Configuring the control register (ADCN) as shown sets the ADC for continuous conversion and for generating an interrupt signal every 16 samples.

Bit #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	–	–	–	–	ADINT 1	ADINT 0	ADCLK 1	ADCLK 0	IREF EN	AD CONT	AD DAIE	AD PM0	ADACQ (4 bits)			
Value	0	0	0	0	1	1	0	0	1	1	1	0	0	0	0	0
Access	r	r	r	r	rw[1]	rw[1]	rw[1]	rw[1]	rw[1]	rw[1]	rw[1]	rw[1]	rw[1]	rw[1]	rw[1]	rw[1]

1 - May only be written when ADCONV=0.

B. Configuration settings for the ADST register.

Bit #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	–	–	–	–	ADDAT 3	ADDAT 2	ADDAT 1	ADDAT 0	REFOK	AD CONV	ADDAI	ADCFG	ADIDX 3	ADIDX 2	ADIDX 1	ADIDX 0
Value	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Access	r	r	r	r	r	r	r	r	r	rw[1]	rw	rw	rw	rw	rw	rw

1 - ADCONV may not be written when PMME=1 and SWB=0.

C. Bits are set to select the MCU's internal reference and conversion channel 0 for the ADC.

Bit #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	–	–	–	–	–	–	–	–	–	ADREF	AD ACQEN	AD ALGN	AD DIFF	ADCH 2	ADCH 1	ADCH 0
Value	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	r	rw[1]	rw[1]	rw[1]	rw[1]	rw[1]	rw[1]	rw[1]

1 - May only be written when ADCONV=0.

D. Configuring the Conversion Sequence Address Register as shown commands the system to read only from ADC configuration register 0 and sets ADC buffer 0 as the first buffer where samples will be stored.

Bit #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	–	–	–	–	SEQSTORE (4 bits)				–	SEQSTART (3 bits)			–	SEQEND (3 bits)		
Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	rw[1]	rw[1]	rw[1]	rw[1]	r	rw[1]	rw[1]	rw[1]	r	rw[1]	rw[1]	rw[1]

1 - May only be written when ADCONV=0.

Figure 5

Click on image to enlarge.

ADDAIE = 1 (An interrupt will be triggered)

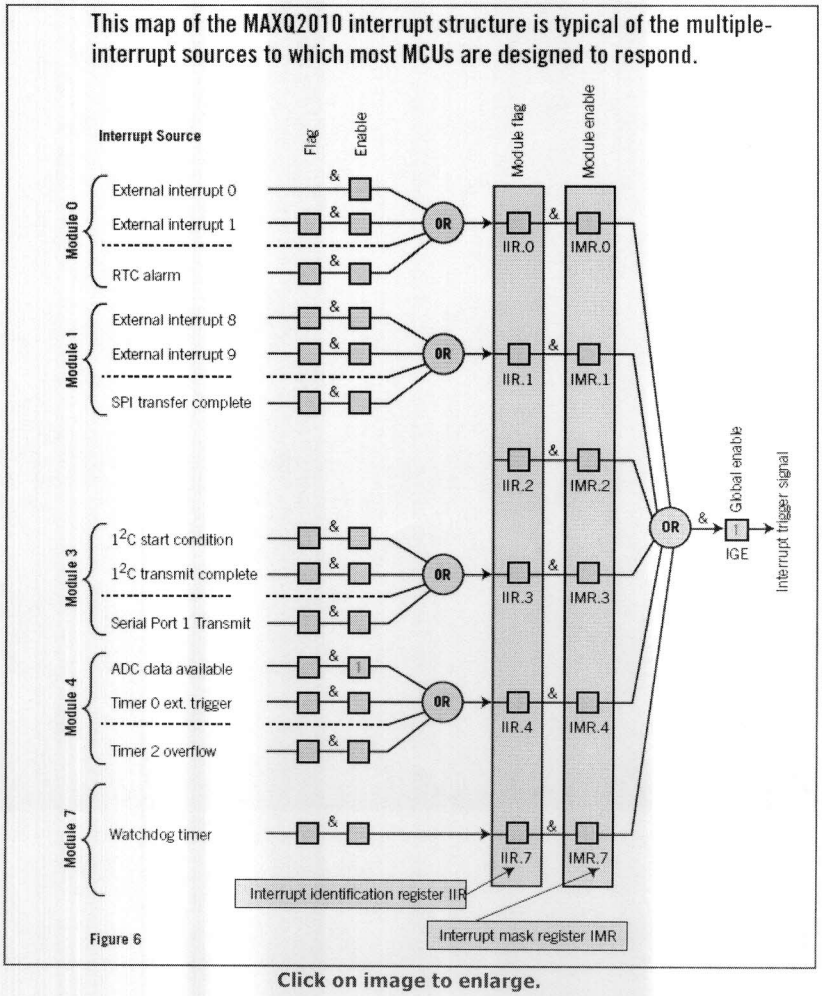
ADCONT = 1 (Continuous conversion sequence mode)

IREFEN = 1 (Internal reference is used)

ADINT[1:0] = 11 (Interrupt every 16 ADC samples)

Additional configuration settings start with the ADCFG bit in the ADST register, which must be set to write to configuration register 0 ADCFG[0] (ADIDX[3:0]=000) (**Figure 5b**). Configuration register ADCFG[0] must also be set to use the MCU's internal reference for the ADC (ADREF=1) and to select conversion channel 0 (ADCH[2:0]=0) in single-ended mode (ADDIFF=0) (**Figure 5c**). Finally, the Conversion Sequence Address Register (ADADDR) must be configured to read ADCFG[0] only, using ADBUF[0] as the first buffer where samples will be stored (**Figure 5d**).

When the converter and MCU begin to capture data, every 16 samples triggers the system to generate an interrupt, which in turn performs the sample averaging. For that purpose, you must also configure the interrupt trigger logic internal to the MAXQ2010. The flow of interrupts is mapped in **Figure 6.**



Click on image to enlarge.

The logical structure begins with the individual interrupt sources listed on the left. Every source has an associated interrupt flag--a special bit set by hardware when the interrupt event from that source is detected. Each source also has an individual interrupt-enable bit that allows applications to enable or disable the interrupt source. The interrupt signal from that source is only active when both the flag and enable bits are set to 1.

The MAXQ MCU's modular architecture allows interrupts to be grouped according to their location in modules (many other MCUs also employ such modular architectures). Like the signals from individual interrupt sources, every module has its own interrupt flag signal and enable bit, as shown in the middle of the figure. The flag is a logical "OR" of all the

underlying interrupt signals from that module, and the enable bit allows applications to enable or disable interrupts for an entire module. These bits are allocated in two 8-bit registers (flags in IIR and enables in IMR), to provide interrupt signaling and control at the module level.

Finally, the interrupt signals from all modules are "OR'd together" to form a global interrupt trigger signal as shown on the right of Figure 6. By enabling or disabling this signal, the interrupt global enable (IGE) bit allows applications to provide interrupt control at the global level. Once set, an individual interrupt flag stays set until cleared by software (in other words, by interrupt service code), even if the condition that caused the flag to set becomes inactive or is removed.

For the ADC peripheral, interrupt control consists of setting/reading the module enable bit, local enable bit, and interrupt flag bit, shown in **Table 1**. Because the ADC is located in module 4, bit 4 of the Interrupt Mask Register (IMR) is set at the end of the initADC function, to enable interrupts for that module.

Interrupt	ADC Data Available Interrupt
Module enable bit	IM4 (IMR.4)
Local enable bit	ADDAIE (ADCN.5)
Interrupt flag	ADDAU (ADST.5)

Table 1

Using the IAR compiler (or equivalent compilers for the MCU you use), you can enable global interrupts by calling the **`__enable_interrupt()`** function. This function sets the Interrupt Global Enable (IGE) bit of the Interrupt and Control (IC) register.

Listing 2

```
// Interrupt handler for ADC occurs when ADDAI is set to "1" from sequencer

#pragma vector = 4
__interrupt void adcInterrupt()
{
ADST_bit.ADCONV = 0;    // stop conversion
}
```

Click on image to enlarge.

Finally, the use of an interrupt requires initializing the interrupt vector. The IAR compiler allows a different interrupt handling function for each module. Setting the interrupt handler for a particular module requires using the #pragma vector directive. The interrupt-handling function declaration should also be preceded by the **`__interrupt`** keyword. The example application declares an interrupt handler for module 4, as illustrated in **Listing 2**. Next, the Status Register (ADST) must be configured to start the conversion:

```
void enable_conversion(){ADST_bit.ADCONV = 1;      //enable conversion}
```

Listing 3

```
// cycle executed waiting interrupt

void idle()
{
int j=0;
while (ADST_bit.ADCONV == 0x1) //interrupt has not be served
yet
   {
   for (j=0; j<1; j++);
   }
}
```

Click on image to enlarge.

With the settings used in the previous figures, 16 samples on channel 0 are collected and stored in ADBUF[15:0], and an interrupt is generated after 16 samples. While waiting for the interrupt to trigger, the system executes an idle function in **Listing 3**. When an interrupt is detected, ADBUF[15:0] can be accessed as follows:

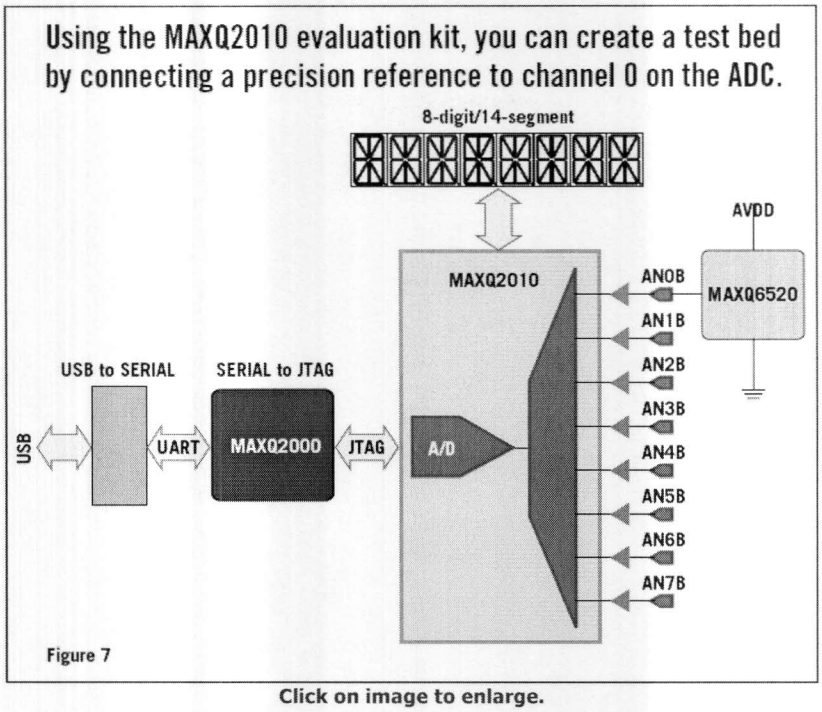
- Status Register (ADST) is configured to read ADBUF[15:0] via ADDATA
- ADCFG = 0
- AIDDX[3:0] = 0000
- Reading from ADDATA returns the value held in ADBUF[0].

Because reading from ADDATA causes the value of AIDDX[3:0] to auto increment, reading ADDATA successively returns the values ADBUF[1], ADBUF[2], ADBUF[3], ADBUF[4], ADBUF[5], ADBUF[6], ADBUF[7], ADBUF[8], ADBUF[9], ADBUF [10], ADBUF[11], ADBUF[12], ADBUF[13], ADBUF[14], and ADBUF[15]. The next step consists of adding the values returned from ADBUF[15:0] and dividing by 16, as shown in **Listing 4**.

Listing 4

```
// Get 16 samples, store them on accumulator and average

float getADCReading()

 {
   unsigned short data=0;
   int i=0;
   static long accumulator=0L;// here we accumulate samples
   float result= 0L;          // result of averaging

   ADST_bit.ADCFG = 0;        // ADBUF reading
   ADST_bit.ADIDX = 0x0;      // start from ADBUF[0]

   while (i<=15)              // read ADBUF
   {
   data = ADDATA;
   accumulator+=data;
   i++;
   }
   result = (accumulator/16.0);    // average
   return result;
 }
```

Click on image to enlarge.

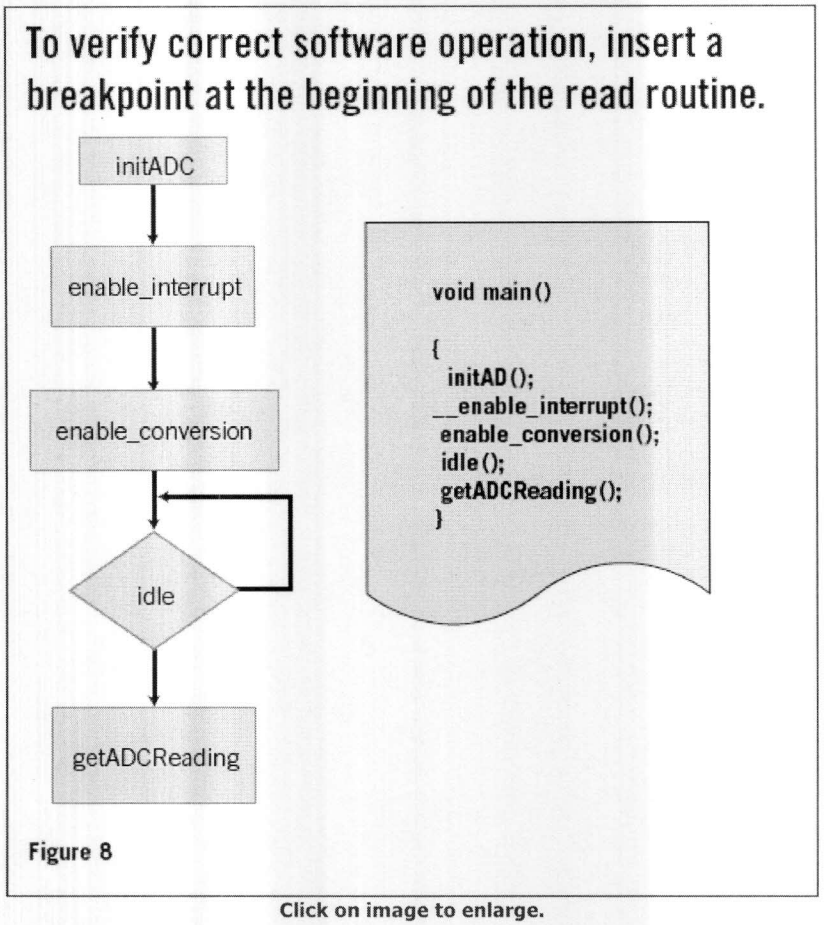


Click on image to enlarge.

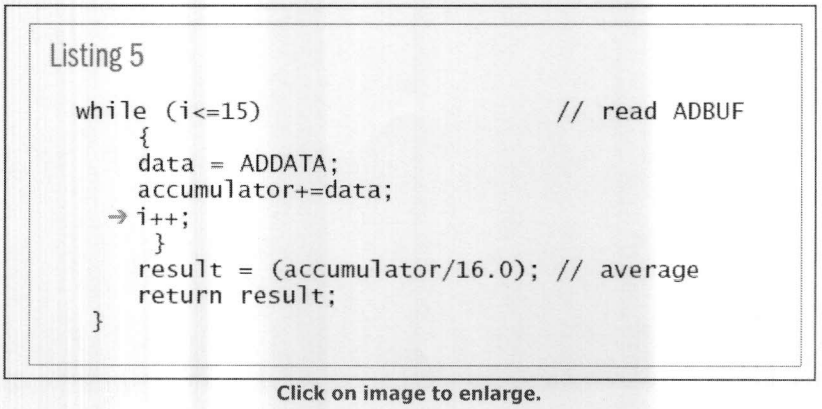
This sample code has been tested on the MAXQ2010 evaluation kit, using the IAR Compiler. A 1.20 V ±1% voltage reference (MAX6520) is connected to channel 0 of the ADC as shown in **Figure 7**, with output voltage guaranteed to remain between 1.176V and 1.224 V over the full operating temperature range. The MAXQ2010 internal reference is 1.5 V, so one LSB = 1.5 V/4096 = 0.36 mV. The expected values are:

- Accumulator from 52256 to 54400 (nominal 53333)
- Result from 3266 to 3400 (nominal 3333).

The main program flow initializes the ADC and then enables the interrupt, starts the conversion sequence, and captures the conversion result (**Figure 8**). The oversampling routine does this 16 times, and saves each result in a buffer. After all 16 samples are collected in the buffer, the processor averages the result to deliver an enhanced-resolution output. To better monitor the operation of this program, you can insert a breakpoint in the program just before the read operation shown in **Listing 5**. Software can then monitor the values of local variables (accumulator, data, i, result) as shown in **Figure 9**.



Click on image to enlarge.



Click on image to enlarge.

The results after the first reading (ADBUF[0], i = 0).

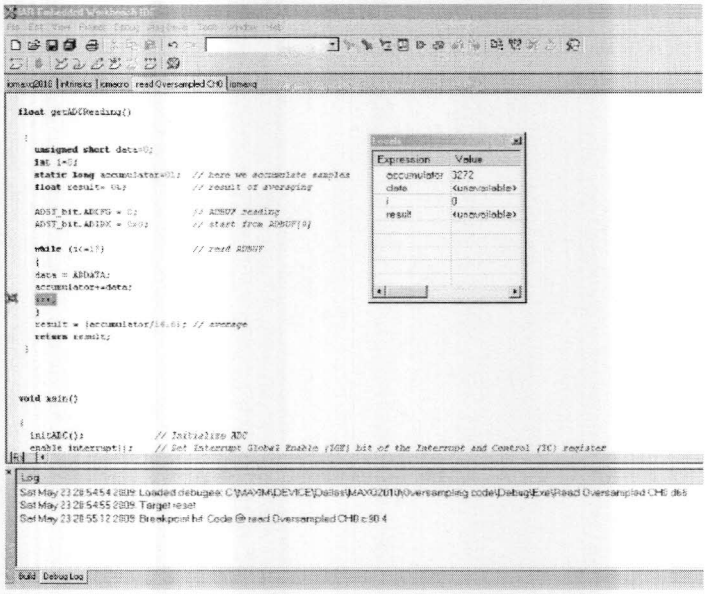


Figure 9

Click on image to enlarge.

Next, execute the code step by step to verify that the value of the local variable "Accumulator" is updated in the right way each time "i" is incremented (**Figure 10**). Finally, the results are averaged to get the higher-resolution result from the oversampling operation (**Figure 11**).

The results after the ADC steps through the last conversion step (last reading ADBUF[15], i = 15).

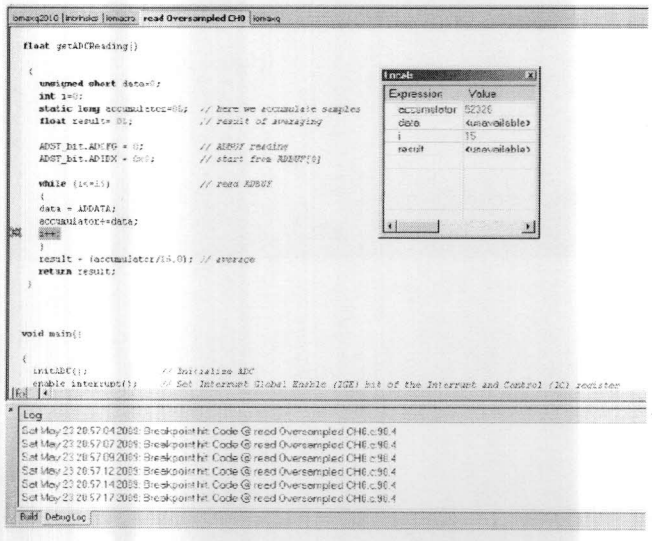


Figure 10

Click on image to enlarge.

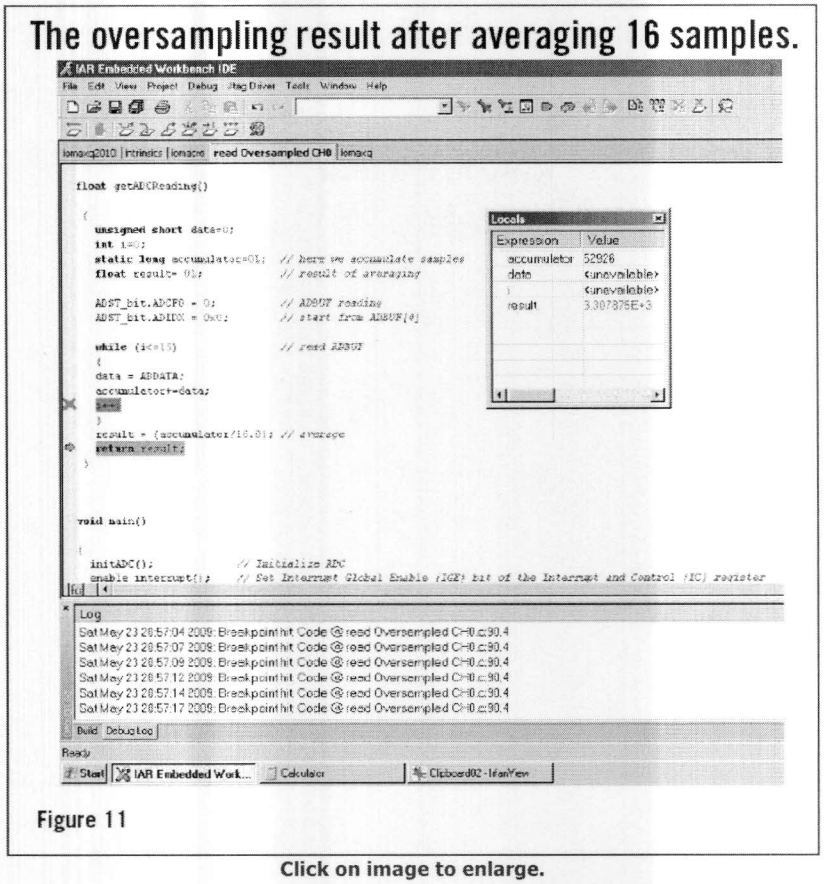


Figure 11

Click on image to enlarge.

Snapshots collected during code debugging show that at the end of the accumulate/average function, the values of the local variables Accumulator and Result were within the expected ranges: Accumulator 52926, Result 3307.875. The measured voltage is therefore 3307.875 x 0.36 mV = 1.190835 V, which is a consistent increase in granularity for a 12-bit system.

Franco Contadini *holds dual roles at Maxim. He is a corporate applications engineer for the Microcontroller Business unit and a principal field applications engineer for OEMs in primarily the telecommunications and medical markets. Prior to joining Maxim in 1998, he worked for military and telecommunications companies, including Marconi, Italtel, and others. He earned a degree in electronics in 1983.*

article

Track This Thread

Sends you email alerts when someone comments on this article